CS-412 Fuzzing Project Report: libpng

Raphaël, Simone, Alexander, Samuel

May 15, 2025

Abstract

1 Introduction

1.1 libpng

For this fuzzing project we chose the libpng library, a widely used C library for reading and writing PNG image files. The oss-fuzz project for libpng only contains one harness, `libpng_read_fuzzer`. It has a code coverage of 43.3%, as seen here.

1.2 Students

The students in our group are:

- Alexander Odermatt, 315842
- Raphaël Küpfer, 283435
- Simone Andreani, 325496
- Samuel Tepoorten, 299798

2 Part 1

Running the fuzzer

As instructed, we ran `libpng_read_fuzzer` once for 4 hours with seeds and once without seeds. In order to run the fuzzer without seeds, the corresponding lines in `./libpng/contrib/oss-fuzz/build.sh` within the oss-fuzz libpng project folder had to be commented out as shown by project.diff. Moreover, for the locally modified changes to be integrated into the build process of the fuzzers, `./Dockerfile` had to be modified as well to copy a local libpng repository previously adjusted instead of importing it directly from the remote libpng repository. These changes can be examined through oss-fuzz.diff. The diff files can be found in the actual folder (part1) or via the links in this report.

Remark : The oss-fuzz.diff has been applied in both cases (seeds, no seeds) to allow us to use a specific version of the libpng commit.

For the hereafter discussed coverage, `run.w_corpus.sh` and `run.w_o_corpus.sh` were respectively used for the fuzzing with seed corpus and without seed corpus (see the README.md for the detailled procedure).

Listing 1: libpng oss-fuzz.diff

```
diff --git a/projects/libpng/Dockerfile b/projects/libpng/Dockerfile
index 6f281cd55..2326c0bc6 100644
--- a/projects/libpng/Dockerfile
+++ b/projects/libpng/Dockerfile
@@ -19,6 +19,7 @@ RUN apt-get update && \
     apt-get install -y make autoconf automake libtool zlib1g-dev

 RUN git clone --depth 1 https://github.com/madler/zlib.git
-RUN git clone --depth 1 https://github.com/pnggroup/libpng.git
+#RUN git clone --depth 1 https://github.com/pnggroup/libpng.git
+COPY libpng_Custom_Repo libpng
 RUN cp libpng/contrib/oss-fuzz/build.sh $SRC
 WORKDIR libpng
```

Listing 2: libpng project.diff

```
diff --git a/contrib/oss-fuzz/build.sh b/contrib/oss-fuzz/build.sh
index 7b8f02639..be8498b72 100755
--- a/contrib/oss-fuzz/build.sh
+++ b/contrib/oss-fuzz/build.sh
@@ -43,8 +43,8 @@ $CXX $CXXFLAGS -std=c++11 -I. \
      -lFuzzingEngine .libs/libpng16.a -lz

 # add seed corpus.
-find $SRC/libpng -name "*.png" | grep -v crashers | \
-     xargs zip $OUT/libpng_read_fuzzer_seed_corpus.zip
+#find $SRC/libpng -name "*.png" | grep -v crashers | \
+#     xargs zip $OUT/libpng_read_fuzzer_seed_corpus.zip

 cp $SRC/libpng/contrib/oss-fuzz/*.dict \
      $SRC/libpng/contrib/oss-fuzz/*.options $OUT/
```

Default vs Empty Corpus

Table 1: Comparison of Line Coverage in Percentages

Path	With Corpus	Without Corpus	Decrease
contrib/	87.93	63.79	27.45
png.c	43.45	26.57	38.85
pngerror.c	55.26	34.47	37.62
pngget.c	3.50	3.50	0.00
pngmem.c	79.28	72.07	9.09
pngread.c	28.36	8.49	70.06
pngrio.c	78.57	78.57	0.00
pngrtran.c	30.17	8.68	71.23
pngrutil.c	73.35	55.71	24.05
pngset.c	53.03	43.18	18.57
pngtrans.c	8.67	4.10	52.71
Totals	41.86	24.98	40.32

With a total decrease of over 40% the coverage data in table 1 shows that the fuzzing run without any provided corpus results in an overall worse line coverage. This is also the case for all files except `pngget.c` and `pngrio.c`, which have equal coverage. The most drastic decrease in line coverage happened for the `pngrtran.c` file which went from 30.17% to only 8.68%. For example in the fuzzing run without the corpus, the function `png_do_scale_16_to_8` in line 2463 in `pngrtran.c` never gets called as `PNG_READ_SCALE_16_TO_8_SUPPORTED` is not defined (if statement in line 4990). These results show that the provided corpus has a positive effect on fuzzing performance.

3 Part 2

Here we identifies two significant uncovered code regions in the `libpng` library with the help of OSS-Fuzz Introspector report generated as follow from the oss-fuzz directory :

```
python3 infra/helper.py introspector libpng
python3 -m http.server 8008 --directory /.../oss-fuzz/build/out
    /libpng/introspector-report/inspector
```

We selected the functions `OSS_FUZZ_png_set_quantize` and `png_handle_iCCP` base on Introspector results and some other parameters discussed thereafter (Subsequently, we decided to take the function `png_handle_iCCP` instead of `png_get_tRNS`, we provide further explanations later on).

`OSS_FUZZ_png_set_quantize` **:**

This function is located in `/src/libpng/pngrtran.c` and we first observe the following interesting things in the introspector results :

- **Hitcount : 0** → The function has not been executed during testing.
- **Instr count : 1626** → This extremely large number suggests that the function is performing many operations. This make it critical for performance and security implications.

Other parameters like basic block count (190), cyclomatic complexity (58) and total cyclomatic complexity (113) are also quite big, indicating that this function has many distinct pathways for execution, so a high potential for bugs and vulnerabilities. Finally, the Unreached Complexity (58) suggests that a significant portion of the function's logic remains untested.

In terms of its usefulness, this function is responsible for configuring the quantization process in PNG files which may be critical because it allows to reduce the number of colors in an image while maintaining its visual integrity. This operation is used in applications where image size optimization is important like web browsers, mobile devices or even embedded systems so it is important to ensure its robustness.

After the analysis of `/libpng/contrib/oss-fuzz/libpng_read_fuzzer.cc`, the existing harness, we notice that there is no call to this function which nevertheless seems important and commonly used. Moreover, the seed corpus likely lacks PNG files with palettes/configurations that trigger it.

`png_handle_iCCP` :

This function is located in `/src/libpng/pngrutil.c` and we first observe the following interesting things in the introspector results :

- **Hitcount : 0** → The function has not been executed during testing.
- **Instrumentation Count : 547** → This moderately high count (The second most significant in the results) reflects, like before, that this function is critical for performance and security implications.

Basic Block Count (70) and Cyclomatic Complexity (28) are smaller than for `OSS_FUZZ_png_set_quantize` but stay significant. In other hand, total cyclomatic complexity (262) is very high and unreachable complexity also (93), there is thus room for improvement.

In terms of its usefulness this function is responsible for processing and managing International Color Consortium profiles embedded in PNG files. Profiles are metadata blocks that define how colors need to be interpreted to ensure their consistence and visual fidelity across various devices. This function is particularly used in workflows where precision is important such as professional imaging, graphic design and printing.

After the analysis of `/libpng/contrib/oss-fuzz/libpng_read_fuzzer.cc`, the existing harness, we notice that there is no call to this function which is not surprising as the function is slightly more specific than others in the same file. Furthermore it is unlikely that an image containing such metadata was included in the seed corpus because this type of images are not common in generic datasets.

RMQ : Unfortunately, we couldn't find a way to reach this function, so we had to opt for the following alternative

`png_get_tRNS` :

Althought png_handle_iCCP would have been more relevant, this one, located in `/src/libpng/pngget.c` has a hit count of **0** and a function line coverage of **0.0%** suggesting that, while relatively simple, it remains unexplored by the current fuzzing methodology.

Despite the other metrics are not so favorable (see Png_get_tRNS_result), the reason to improve the coverage of this function is that it has an important role in transparency management (via the `tRNS` chunk). This enables defining transparency in indexed-color images, truecolor images or single transparent colors for grayscale for example and while modern image processing workflows often use alpha channels, the `tRNS` chunk remains a part of the PNG specification and is widely used in lightweight images.

We conclude that with these observations, it remains interesting to cover this function.

Analyses and suggestions

Optimal target analysis

Remaining optimal interesting functions

The following table shows a list of functions that are optimal targets. Optimal targets are identified by finding the functions that in combination, yield a high code coverage.

Columns Rows Search table

Func name	Functions filename	Function depth	hitcount	instr count	bb count	cyclomatic complexity	Reachable functions	Incoming references	total cyclomatic complexity	Unreached complexity
OSS_FUZZ_png_check_keyword	/src/libpng/pngset.c	3	0	246	46	17	7	0	65	39
OSS_FUZZ_png_read_png	/src/libpng/pngread.c	11	0	384	67	22	132	0	1318	46
OSS_FUZZ_png_set_cHRM_XYZ	/src/libpng/pngset.c	6	0	74	3	2	14	0	70	39
OSS_FUZZ_png_set_quantize	/src/libpng/pngrtran.c	6	0	1626	190	58	14	0	113	58
OSS_FUZZ_png_set_sCAL	/src/libpng/pngset.c	5	0	67	9	4	18	0	124	81
png_handle_iCCP	/src/libpng/pngrutil.c	9	0	547	79	28	41	0	262	93
png_handle_iTXt	/src/libpng/pngrutil.c	8	0	499	83	29	36	0	239	87
png_handle_pCAL	/src/libpng/pngrutil.c	8	0	498	64	23	27	0	159	43

Showing 1 to 8 of 8 entries
Previous 1 Next

Figure 1: Introspector results

Func name	Functions filename	Reached by Fuzzers	Func lines hit %	Cyclomatic complexity	Functions reached	Reached by functions	Accumulated cyclomatic complexity	Undiscovered complexity
OSS_FUZZ_png_get_tRNS	/src/libpng/pngget.c	0	0.0%	10	0	0	10	10

Showing 1 to 1 of 1 entries (filtered from 401 total entries)
Previous 1 Next

Figure 2: Introspector Png_get_tRNS result

4 Part 3

Overview

In this section we describe the improvements made to the fuzzing infrastructure to increase the coverage of two previously uncovered code regions identified in Part 2:

- `png_set_quantize` in `pngrtran.c`
- `png_get_tRNS` in `pngget.c`

We explain the rationale for our implementation, the specific changes made to the harness, the use of seeds, the build scripts, and we present the results obtained after running the modified harnesses for 4 hours.

4.1 Setup

In order to properly conduct fuzzing while taking into account the changes applied to the harness and possible custom seeds, we created a `run.improve.sh` script. This script initially clones the correct oss-fuzz and libpng repositories via `cloneRepositories()`. Then, it adds the custom seeds originally placed in the folder custom_seeds to the libpng folder via `addNewSeeds()`. Next, it applies the correct changes present in the `oss-fuzz.diff` and `project.diff` files via `applyDiff()`, and initializes important directories thanks to `initDirectories()`. Then, the fuzzers are built and executed using `buildFuzzers()` and `runFuzzers()`, and finally, the coverage report is generated using `createCoverageReport()`. This structure allows us to easily adapt the script to different cases. For instance, if you don't need to use custom seeds to fuzz a particular function, you can modify the script by skipping the custom_seeds folder.

4.2 Improvement 1: Covering `png_set_quantize`

Implementation

To cover this region, we modified the existing harness (`libpng_read_fuzzer.cc`) by directly calling `png_set_quantize`. To correctly call the function, it is essential to provide a valid color palette as argument, which is contained in a specific chunk of the PNG image, called PLTE. To retrieve it we used the libpng function `png_get_PLTE`, and if the PNG image contained a valid PLTE chunk, we would provide the palette to `png_set_quantize`. This function also accepts an optional histogram as argument, which is used to help quantize the image. As with the palette, the histogram is contained in a specific chunk called hIST, and can be retrieved through the function `png_get_hIST`.

The modifications to the harness proved to be beneficial in terms of increasing coverage, which however did not extend to the entire function. Indeed, the fuzzer never covered the part of the function inside `"if (histogram != NULL)"`, which meant that the current seeds were not the best ones. After reviewing the chunks of the initial seeds from the libpng repository through a command shown in Listing 3, we determined that in fact none of the images contained a hIST chunk. Therefore, we proceeded to incorporate new custom seeds to the fuzzer. To do so, we created two images using ImageMagick, and through a custom script we added a histogram to each one. This would theoretically allow us to get into the uncovered part of the code, however due to an unknown mistake on our side, the fuzzer didn't manage to enter that part. This could be due to the way we insert the histogram into the NPC, which may not be fully compatible with the target function, or an error in the harness itself.

Listing 3: Output of `pngcheck -v [FILENAME].png`

```
$ pngcheck -v histed.png
File: histed.png (309 bytes)
  chunk IHDR at offset 0x0000c, length 13
    4 x 1 image, 8-bit palette, non-interlaced
  chunk gAMA at offset 0x00025, length 4: 0.45455
  chunk cHRM at offset 0x00035, length 32
    White x = 0.3127 y = 0.329,  Red x = 0.64 y = 0.33
    Green x = 0.3 y = 0.6,  Blue x = 0.15 y = 0.06
  chunk PLTE at offset 0x00061, length 6: 2 palette entries
  chunk hIST at offset 0x00073, length 4: 2 histogram entries
  chunk tIME at offset 0x00083, length 7: 14 May 2025 15:47:04 UTC
  chunk caNv at offset 0x00096, length 16
    unknown private, ancillary, safe-to-copy chunk
  chunk IDAT at offset 0x000b2, length 13
    zlib: deflated, 256-byte window, maximum compression
  chunk tEXt at offset 0x000cb, length 37, keyword: date:create
  chunk tEXt at offset 0x000fc, length 37, keyword: date:modify
  chunk IEND at offset 0x0012d, length 0
No errors detected in histed.png (11 chunks, ~7625.0% compression).
```

Due to this problem, we had to abandon the idea of using custom seeds and instead focused on modifying the harness to allow for increased overall coverage. As shown in Listing 4, in cases where the seed did not have an hIST chunk, the code creates an histogram populated with random elements. To allow for the possibility of a null histogram, the mock histogram is only created if the condition `"size %2 == 0"` is true. This is not the optimal solution, but still allows us to have good coverage of the function.

Listing 4: `png_set_quantize` call placed into `libpng_read_fuzzer.cc`

```
  if ((color_type & PNG_COLOR_MASK_COLOR) != 0) {
    int num_palette;
    png_colorp palette;

    if (png_get_PLTE(png_handler.png_ptr, png_handler.info_ptr, &palette, &num_palette) != 0) {
        png_uint_16p histogram = NULL;

        png_get_hIST(png_handler.png_ptr, png_handler.info_ptr, &histogram);

        png_uint_16 default_hist[256];
        if (histogram == NULL && num_palette > 0 && num_palette <= 256 && size % 2 == 0) {
          for (int i = 0; i < num_palette; ++i) {
            default_hist[i] = (png_uint_16)(rand() % 65536);
          }
          histogram = default_hist;
        }

```

```
        if (color_type == PNG_COLOR_TYPE_PALETTE) {
          png_set_quantize(png_handler.png_ptr, palette,
              num_palette, 255, histogram, 0);
        } else {
          png_set_quantize(png_handler.png_ptr, palette,
              num_palette, 255, histogram, 1);
        }
    }
  }
```

Evaluation

We ran the modified harness for 4 hours and generated the coverage report. Compared to the original version, we observed a substantial increase in the coverage of `pngrtran.c` and an almost complete coverage of the previously uncovered `png_set_quantize` function. The total library coverage increased from 41.83% before the modifications to 44.06%; the `pngrtran.c` coverage increased from 30.17% with 1025/3397 lines covered, to 40.39% with 1372/3397 lines. The almost entirety of `png_set_quantize` is now covered in the introspection report. It is certainly possible to further improve the coverage of the feature, for example by creating seeds with specific color combinations and containing certain chunks such as histogram.

4.3 Improvement 2: Covering `png_get_tRNS`

Implementation

To trigger the execution of `png_get_tRNS`, we modified the fuzzing harness to explicitly call the function. In fact, this function was not covered by the fuzzer because it is not reachable from the harness. To correctly call it we needed to provide as argument the png details, and three pointers: *trans_ alpha*, *num_ trans*, and *trans_ color*; as seen in 5.

Listing 5: `png_get_tRNS` call placed into `libpng_read_fuzzer.cc`

```
png_bytep trans_alpha = nullptr;
int num_trans = 0;
png_color_16p trans_color = nullptr;

png_get_tRNS(png_handler.png_ptr, png_handler.info_ptr,
             &trans_alpha, &num_trans, &trans_color);
```

The creation of custom seeds was not necessary for this function, unlike the first one. Therefore we were able to proceed by modifying the harness.

Evaluation

After running the fuzzer for 4 hours, we observed that the total library coverage increased from 41.83% before the modifications to 42.09%; the `pngget.c` coverage increased from 3.50% with 26/742 lines covered, to 8.09% with 60/742 lines. The entirety of `png_get_tRNS` is now covered in the introspection report. The coverage was achieved by modifying the harness and without any manually crafted seeds.

4.4 Notes

To select the second function to be fuzzed, we conducted extensive research and performed numerous tests on many other functions. However, they often proved to be unsuitable. A large portion of the functions with low coverage were set as private, so it would have been impossible to fuzz them without modifying the library directly. Additionally, we looked for functions with arguments that depended as much as possible on the PNG input so that the fuzzer could work more effectively on seeds. However, most of the functions had arguments that were independent of the PNG input and given by the user. Consequently, we were left with relatively simple functions that required only a change in harness to be fuzzed entirely. Therefore, there was no need to create custom seeds.

5 Part 4

Since we didn't discover any new bug, we reproduce here a previous memory corruption crash.

The chosen bug is the following : CVE-2019-7317 (patched from libpng 1.6.37) and affects the `png_image_free` function in the libpng library which is part of the Advanced API introduced in libpng version 1.6.0 and is located in the `png.c` source file.

This function ensures that memory associated with `png_image` objects is properly freed to avoid resource leaks. Therefore, it is a critical component in applications like embedded systems where memory efficiency is important.

The problem with this function is a use-after-free error which occurs specifically when an error is detected in an image being processed. During error handling, the `png_image_free` function is invoked to release resources associated with the `png_image` structure but due to the way `png_safe_execute` calls `png_image_free_function`, memory that has already been deallocated may be accessed again.

The criticality of CVE-2019-7317 is generally considered as medium but use-after-free error can have severe impacts such as application crashes, data corruption or arbitrary code execution. The risk is hight because libpng can processe untrusted externally provided images and is widely used.

To trigger the bug we generate a buggy image with the help of `seed_generator.py` (inspired by public libpng issue) and change the version of libpng to 1.6.36.

The script can be used in the following manner :

- First possibility, reproduce the crash. The script `run.poc.sh` can be run without modifications.
- Second possibility, trigger a new crash. The modification in `run.poc.sh` should be the following (uncomment runFuzzers and comment reproduce) :

```
main() {
  cloneRepositories
  addNewSeeds
  applyDiff
  initDirectories
  buildFuzzers
  runFuzzers
  #reproduce
}
```

To path this vulnerability, the solution is to no use `png_safe_execute` anymore. This mechanism introduces unnecessary complexity by invoking `png_image_free_function` indirectly.

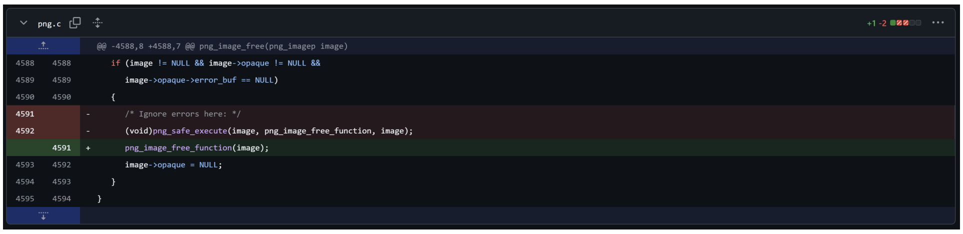

Figure 3: Patch CVE-2019-7317

We executed `run.poc.sh` using a recent version of libpng which includes the patch and confirmed that the bug is no longer triggered.

To execute it :

- Uncomment : `git checkout ea12796`
- Comment : `git checkout eddf902`

- Uncomment : `runFuzzers`
- Comment : `reproduce`

```
cloneRepositories() {
  ...
  git checkout ea12796
  # buggy version of libpng for the POC.
  # git checkout eddf902
  popd
}
...
main() {
  cloneRepositories
  addNewSeeds
  applyDiff
  initDirectories
  buildFuzzers
  runFuzzers
  #reproduce
}
```